Northeastern University

DS 5220 – Lecture 4
Logistic Regression

Roi Yehoshua

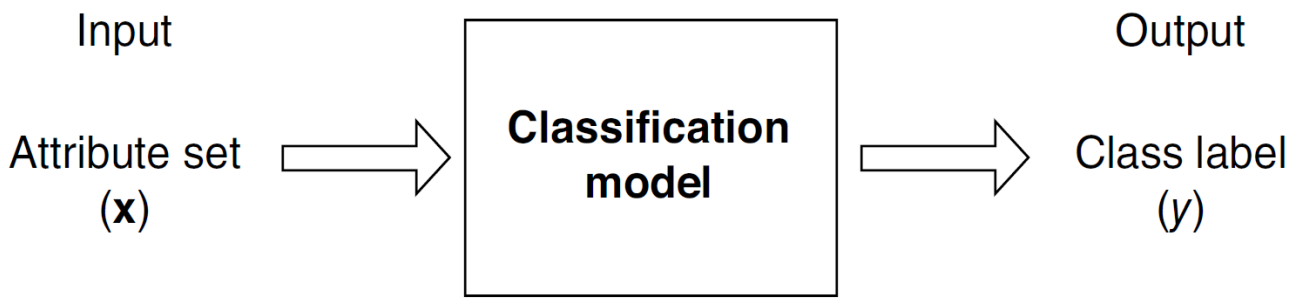
Agenda

Northeastern University

- Classification
- Logistic regression
- The class imbalance problem
- Classification metrics
- Multiclass problems
- Softmax regression

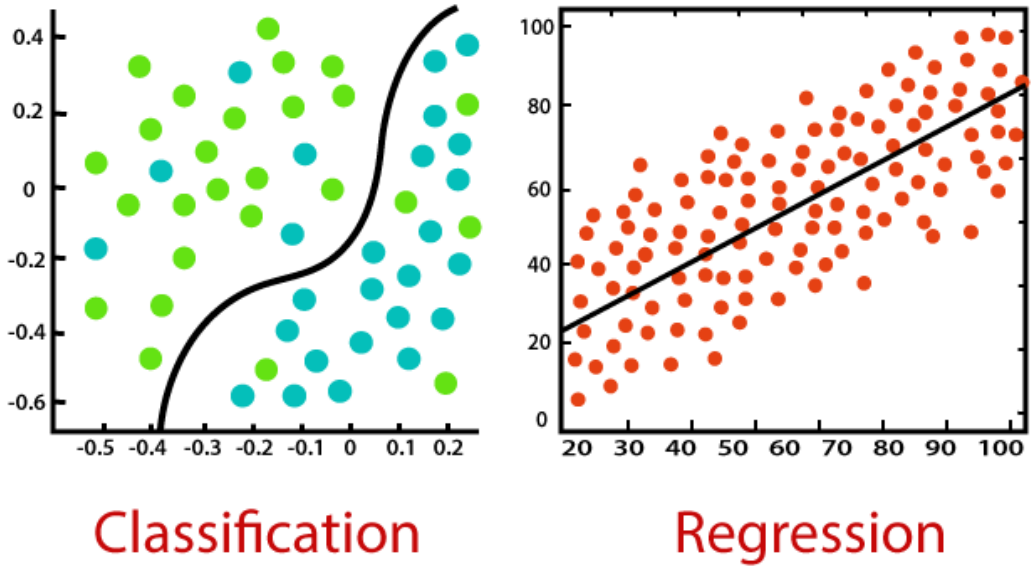
Classification

- **Classification** is the task of predicting the **class** of a given sample
- **Given**: a training set of n labeled examples $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
  - Each label y_i belongs to one of K predefined classes $C_1, \ldots, C_K$
- **Goal**: learn a function $h(\mathbf{x})$ that maps the feature set $\mathbf{x}$ to one of the classes C_i



Classification vs. Regression

- In regression our goal was to find a regression line (hyperplane) that fits the data
- In classification our goal is to find the line (surface) that separates between points that belong to different classes
- This surface is called the **decision boundary/surface**



Classification Tasks Examples

Northeastern University

Task	Feature Set	Class labels
Spam filtering	Features extracted from the email message header and content	Spam or non-spam
Tumor identification	Features extracted from MRI scans	Malignant or benign
Fraud detection	Transaction details such as amount, merchant, location, time	Legit or fraudulent
OCR	Images of text	Characters
Medical diagnosis	Symptoms of the patient	Diseases
Automatic essay grading	Document	Grades

Types of Classification Problems

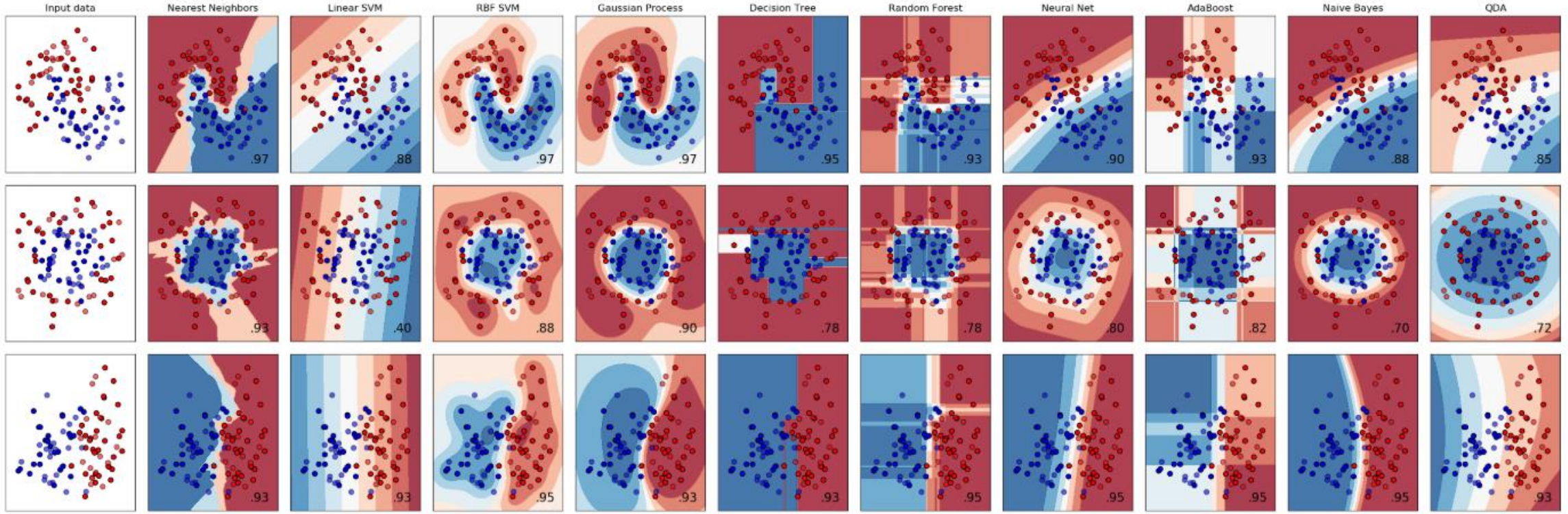
Northeastern University

- In **binary classification** there are only two classes, typically denoted by 1 and 0
  - 1 is called the **positive class** and 0 is called the **negative class**
  - The positive class is usually the class we are more interested in predicting correctly
    - e.g., the “spam” category in email classification problems
- In **multiclass classification** there are more than two possible classes
  - e.g., in image classification the object in the image can be a dog, car, box, person, etc.
  - Some classifiers are designed to deal with binary classes only
  - But can be adapted to deal with multiclass problems (more on this later)
- In **multilabel classification** more than one label may be assigned to each sample
  - e.g., in document classification, we can label each document with all the categories it belongs to

Classification Algorithms

Northeastern University

- There are hundreds of classification algorithms, each with its own pros and cons



http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

Deterministic vs. Probabilistic Classifiers

Northeastern University

- A **deterministic classifier** outputs a categorical class directly
- A **probabilistic classifier** outputs a continuous score between 0 and 1 to indicate how likely is that the sample belongs to a specific class
  - The probability scores for all the classes sum up to 1
  - A sample is typically assigned to the class with the highest probability score
- Probabilistic classifiers provide additional information about the confidence of the classifier in assigning a sample to a class

Logistic Regression

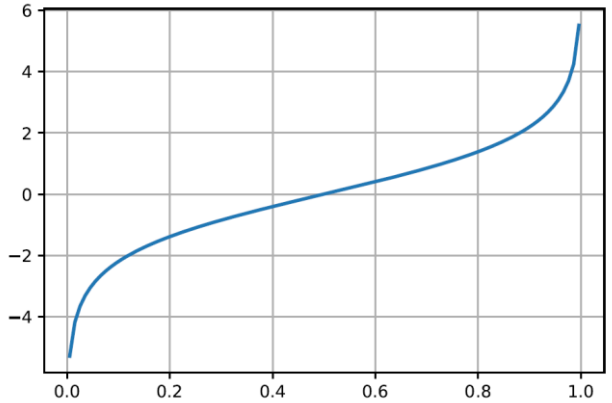
Northeastern University

- A probabilistic classifier that handles binary classification problems
- Given a sample (**x**, *y*), outputs a probability *p* that it belongs to the positive class:

$$p = P(y = 1|\mathbf{x})$$

- If $p > 0.5$, the sample is classified as 1, otherwise it is classified as 0
- The **odds ratio** is the ratio between *p* and 1 – *p*
- **Log odds** (logit) is the logarithm of the odds ratio:

$$\text{logit}(p) = \log \frac{p}{1-p}$$



Logistic Regression

Northeastern University

- Main assumption: **the log odds is a linear combination of the features**

$$\log \frac{p}{1-p} = w_0 + w_1 x_1 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

  - w_0 is the bias as in linear regression
- Solving for p gives us:

$$\frac{p}{1-p} = e^{\mathbf{w}^T \mathbf{x}}$$

$$p = (1-p) e^{\mathbf{w}^T \mathbf{x}}$$

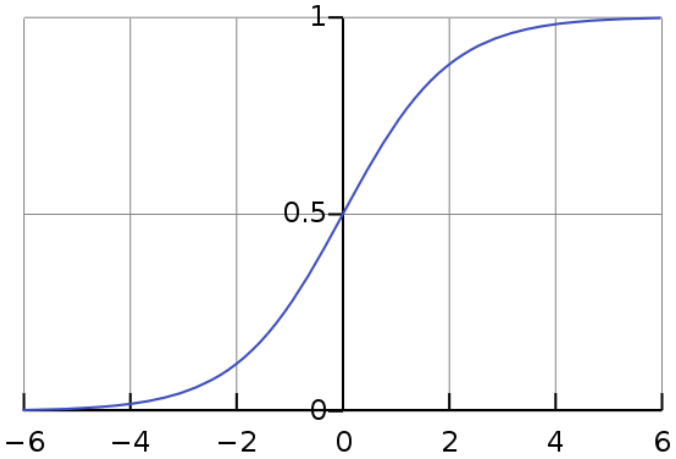
$$p(1 + e^{\mathbf{w}^T \mathbf{x}}) = e^{\mathbf{w}^T \mathbf{x}}$$

$$p = \frac{e^{\mathbf{w}^T \mathbf{x}}}{(1 + e^{\mathbf{w}^T \mathbf{x}})} = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} = \sigma(\mathbf{w}^T \mathbf{x})$$

The Logistic Function

- The logistic function (also called **sigmoid function** due to its S-shape) is:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



- The output of this function can be interpreted as the **probability** of belonging to the positive class
- The function has some convenient mathematical properties such as:

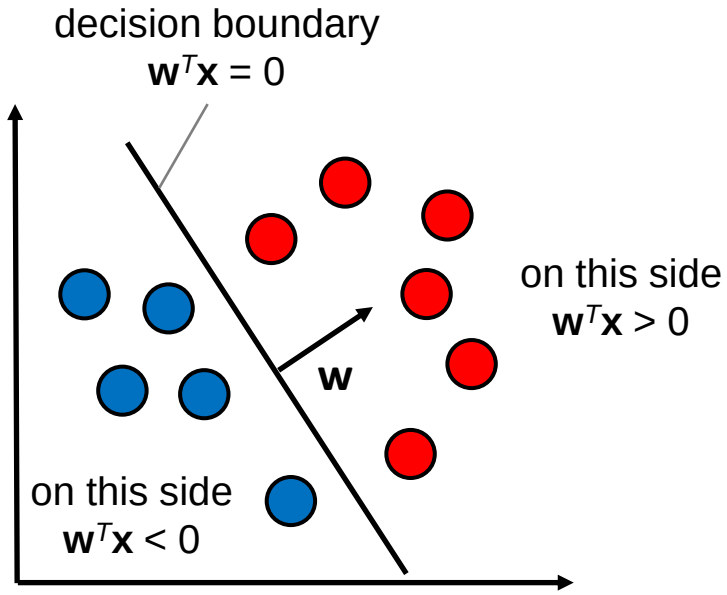
$$\sigma(-z) = 1 - \sigma(z)$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Logistic Regression

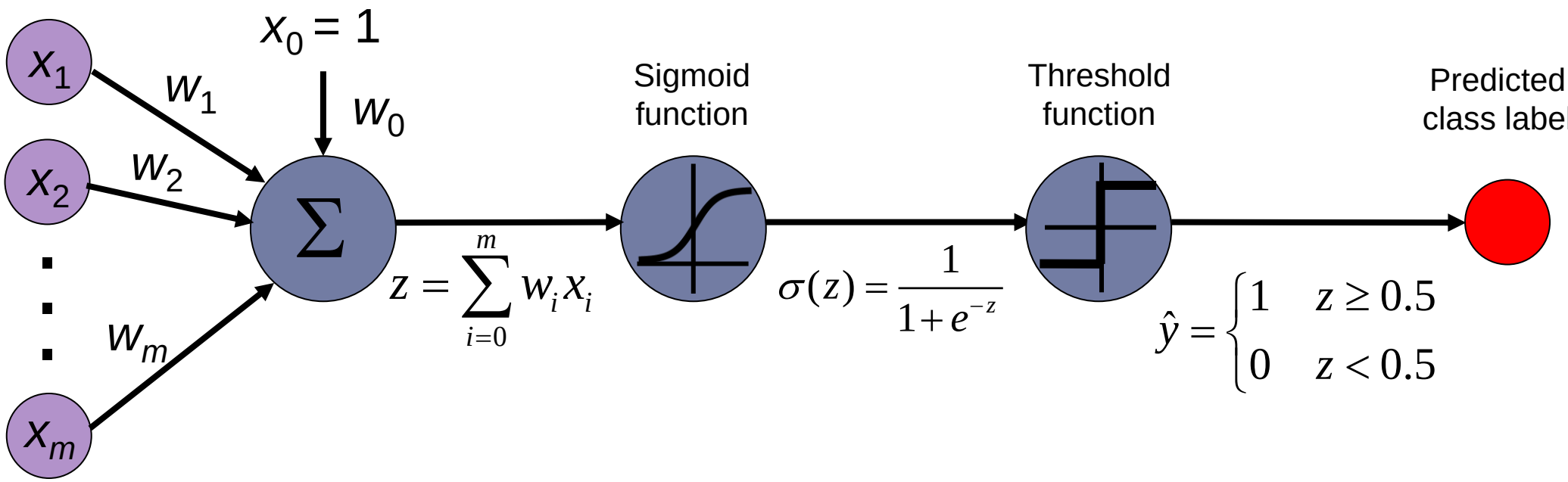
Northeastern University

- The decision boundary is defined by the points for which p = 0.5 ($\mathbf{w}^T\mathbf{x} = 0$)
  - This is a hyperplane of dimension (m - 1)



- An input vector **x** is assigned to class 1 if $\mathbf{w}^T\mathbf{x} \geq 0$ and to class 0 otherwise
- This makes logistic regression a **linear classifier**

Model Summary



$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$$

Example

Northeastern University

- Predict the likelihood of choosing to bike to work based on the temperature

Temperature (°C)	Bikes to work
10	0
15	0
20	1
25	1
30	1

- The logistic regression model $P(y=1|x) = \frac{1}{1+e^{-(w_0+w_1 x)}}$
- Assume that after optimization we found the coefficients w_0 = -12.94, w_1 = 0.74

$$P(y=1|x) = \frac{1}{1+e^{-(-12.94+0.74x)}}$$

- We can use it for predictions: $P(y=1|x=10) = \frac{1}{1+e^{-(-12.94+0.74\cdot 10)}} = 0.0039$

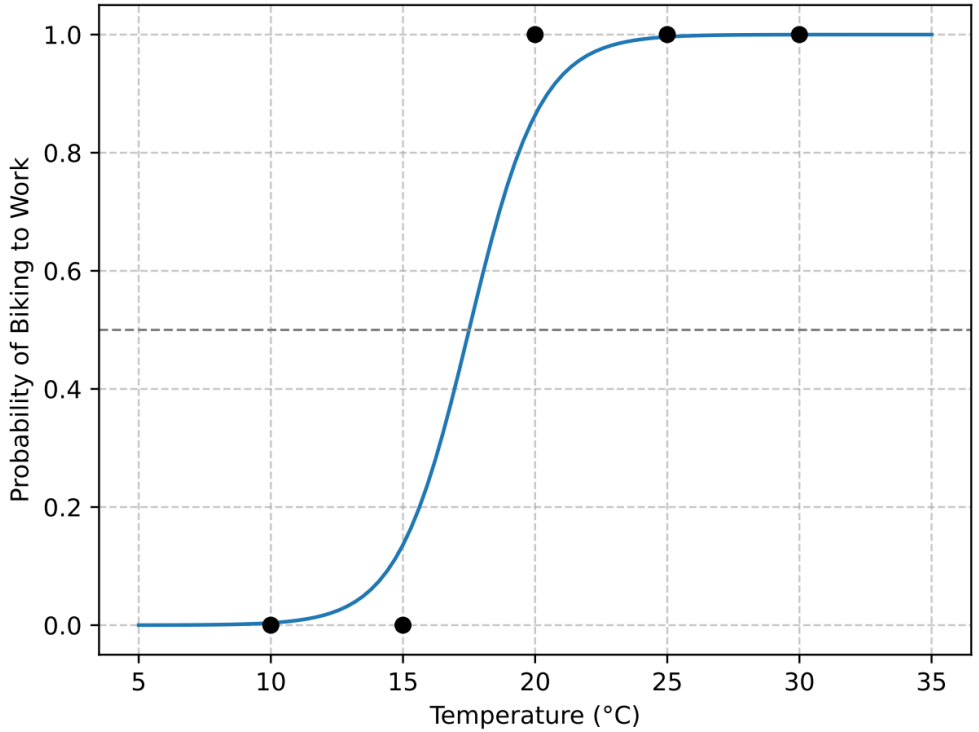
$$P(y=1|x=20) = \frac{1}{1+e^{-(-12.94+0.74\cdot 20)}} = 0.8640$$

$$P(y=1|x=25) = \frac{1}{1+e^{-(-12.94+0.74\cdot 25)}} = 0.9961$$

Example

- The curve of the model

$$P(y = 1|x) = \frac{1}{1 + e^{-(-12.94+0.74x)}}$$



Log Loss

Northeastern University

- Given the logistic regression model, how do we fit **w** for it?
- We need to find a suitable loss function (why can't we use least squares?)
- Again, we can use MLE to derive a loss function
- We assume that the data (the labels) has a Bernoulli distribution with parameter *p*

$$P(y = 1|p) = p$$

$$P(y = 0|p) = 1 - p$$

  - where $p = \sigma(\mathbf{w}^T\mathbf{x})$
- We can write these two equations more compactly as follows:

$$P(y|p) = p^y(1 - p)^{1-y}$$

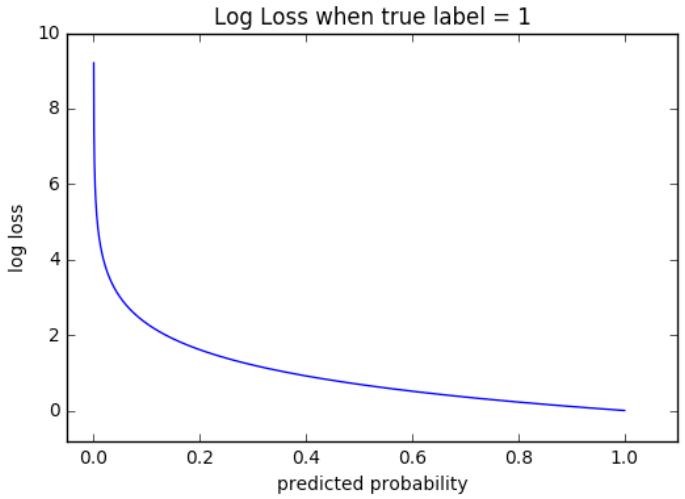
- Therefore, the log likelihood of the data is:

$$\log P(y|p) = y \log p + (1 - y) \log(1 - p)$$

Log Loss

- Taking the negative of the log likelihood we get the log loss function:

$$L_{\log}(y, p) = -y \log p - (1 - y) \log(1 - p)$$



Log Loss

Northeastern University

- The cost function calculates the average loss over the whole data set:

$$J(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

- In vectorized form:

$$J(\mathbf{w}) = -\frac{1}{n}\left(\mathbf{y}^T \log \mathbf{p} + (\mathbf{1}-\mathbf{y}^T)\log(\mathbf{1}-\mathbf{p})\right)$$

- $J(\mathbf{w})$ is a convex function, so it has a single global minimum
- However, there is no closed form solution for the optimal $\mathbf{w}^*$
- Therefore, we need to use iterative approaches like gradient descent

Gradient Descent for Logistic Regression

Northeastern University

- What is the gradient of the log loss function?
- For a single training example (**x**, *y*) we have:

$$
\begin{aligned}
-\frac{\partial}{\partial w_j} J(\mathbf{w}) &= \frac{\partial}{\partial w_j} \left(y \log(h_{\mathbf{w}}(\mathbf{x})) + (1-y) \log(1 - h_{\mathbf{w}}(\mathbf{x})) \right) \\
&= \frac{\partial}{\partial w_j} \left(y \log(\sigma(\mathbf{w}^T\mathbf{x})) + (1-y) \log(1 - \sigma(\mathbf{w}^T\mathbf{x})) \right) \\
&= \left(\frac{y}{\sigma(\mathbf{w}^T\mathbf{x})} - \frac{1-y}{1-\sigma(\mathbf{w}^T\mathbf{x})} \right) \frac{\partial}{\partial w_j} \sigma(\mathbf{w}^T\mathbf{x}) \\
&= \left(\frac{y}{\sigma(\mathbf{w}^T\mathbf{x})} - \frac{1-y}{1-\sigma(\mathbf{w}^T\mathbf{x})} \right) \sigma\left(\mathbf{w}^T\mathbf{x}\right) \left(1 - \sigma(\mathbf{w}^T\mathbf{x})\right) \frac{\partial}{\partial w_j} \mathbf{w}^T\mathbf{x} \\
&= \left(y \left(1 - \sigma(\mathbf{w}^T\mathbf{x})\right) - (1-y)\sigma\left(\mathbf{w}^T\mathbf{x}\right) \right) x_j \\
&= (y - h_{\mathbf{w}}(\mathbf{x}))x_j
\end{aligned}
$$

We used the fact that $\sigma'(z) = \sigma(z)(1- \sigma(z))$

- This gives us the stochastic gradient descent update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$$

Which rule does it remind you?

Regularization in Logistic Regression

Northeastern University

- Similar to linear regression, we can add a regularization term to the log loss function
- L2 regularized logistic regression minimizes the following cost function:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \mathbf{w}^T \mathbf{w}$$

- L1 regularized logistic regression minimizes the following cost function:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda ||\mathbf{w}||_1$$

- Elastic-net is a combination of L1 and L2 penalties

Logistic Regression in Scikit-Learn

Northeastern University

- **LogisticRegression** implements logistic regression using numerical solvers

class `sklearn.linear_model.` **`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) [source]

Parameter	Description
penalty	'l1' or 'l2', default: 'l2'. Used to specify the norm used in the penalization.
tol	Tolerance for stopping criteria
C	Inverse of regularization strength
solver	Algorithm to use in the optimization problem: {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'} • For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones • The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization, or no regularization
max_iter	Maximum number of iterations taken for the solvers to converge
multi_class	{'auto', 'ovr', 'multinomial'}, default='auto' If the option chosen is 'ovr', then a binary problem is fit for each label

Logistic Regression in Scikit-Learn

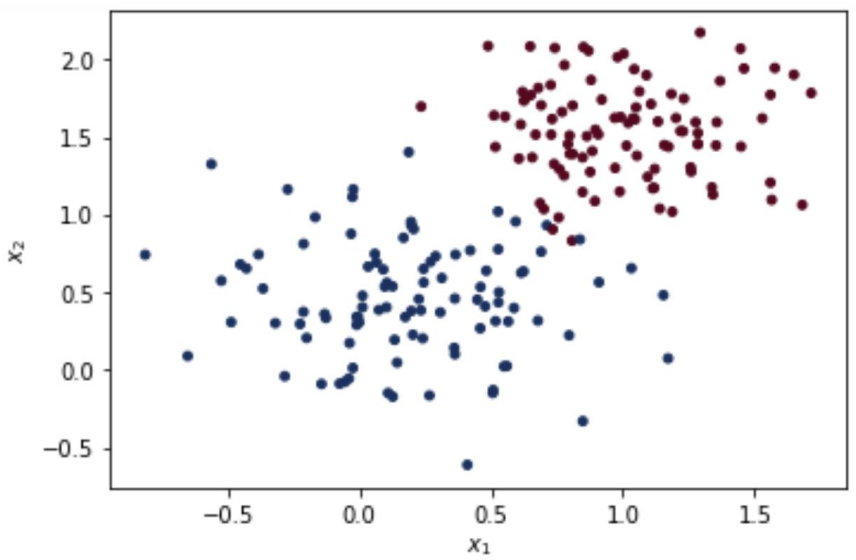
Northeastern University

- We'll first demonstrate logistic regression on a synthetically generated data set
- We'll use the function make_blobs() to generate two normally-distributed blobs with 100 points each

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=200, centers=((1, 1.5), (0.25, 0.5)),
                  cluster_std=(0.3, 0.4), random_state=0)
```

```
def plot_data():
    plt.scatter(X[:, 0], X[:, 1], c=y, s=15, cmap='RdBu')
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
```

```
plot_data()
```



- The blue points belong to class 1 and the orange points belong to class 0

Logistic Regression in Scikit-Learn

Northeastern University

- We now fit a logistic regression model to the data:

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf.fit(X, y)
```

```
LogisticRegression()
```

- We can use the **predict()** method to predict class labels for new samples:

```
X_test = np.array([[1, 1.5], [0, 0], [1, 1]])
clf.predict(X_test)
```

```
array([0, 1, 0])
```

- We can also get the probability estimates for each class by calling **predict_proba()**:

```
clf.predict_proba(X_test)
```

```
array([[0.9534007 , 0.0465993 ],
       [0.00281623, 0.99718377],
       [0.73765835, 0.26234165]])
```

Logistic Regression in Scikit-Learn

Northeastern University

- The **score()** function returns the **accuracy** of the classifier on the given data

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

```
clf.score(X, y)
```

```
0.985
```

Decision Boundary

Northeastern University

- For 2D problems, it is often useful to visualize the boundary line between the classes
- The boundary line is defined by the points for which

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) = 0.5$$

- Hence

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

  - The slope of the line is $-w_1/w_2$ and the intercept is $-w_0/w_2$

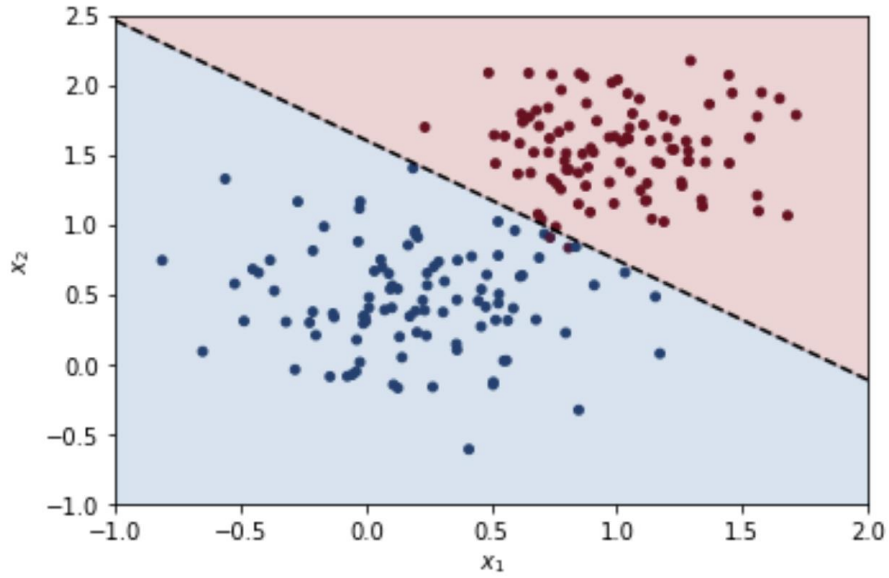
Decision Boundary

Northeastern University

- Plotting the decision boundary:

```
def plot_decision_boundary(clf):
    # Retrieve the model parameters
    w0 = clf.intercept_
    w1, w2 = clf.coef_[0]

    # Calculate the intercept and slope of the decision boundary
    b = -w0 / w2
    m = -w1 / w2

    x_min, x_max = -1, 2
    y_min, y_max = -1, 2.5
    x_d = np.array([x_min, x_max])
    y_d = m * x_d + b

    plot_data()
    plt.plot(x_d, y_d, 'k', ls='--')
    plt.fill_between(x_d, y_d, y_min, color='tab:blue', alpha=0.2)
    plt.fill_between(x_d, y_d, y_max, color='tab:orange', alpha=0.2)
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max);
```

```
plot_decision_boundary(clf)
```



The Class Imbalance Problem

Northeastern University

- Data sets with imbalanced class distribution (e.g., when # of positive examples < 5%) are common in real-world applications
  - fraud detection, spam detection, diagnosis of diseases, …
- Challenges with imbalanced data sets:
  - Metrics such as accuracy may not be meaningful
    - e.g., if 1% of the credit card transactions are fraudulent, a model that predicts every transaction as legitimate has an accuracy of 99%
  - Learning may not focus on the rare class examples at all
    - Models that describe the rare class tend to be highly specialized and are often sensitive to noise
- **A correct classification of the rare class often is more important than a correct classification of the majority class**

Example: Bank Marketing

Northeastern University

- The data set comes from the UCI Machine Learning repository
- Deals with marketing campaigns (phone calls) of a Portuguese banking institution
- The goal is to predict whether the client will subscribe (1/0) to a term deposit
- The data set includes 41,188 records and 21 features
- Information about the data set can be found here

Example: Bank Marketing

Northeastern University

Attribute Information:

Input variables:
bank client data:
1 - age (numeric)
2 - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown')
3 - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
4 - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
5 - default: has credit in default? (categorical: 'no','yes','unknown')
6 - housing: has housing loan? (categorical: 'no','yes','unknown')
7 - loan: has personal loan? (categorical: 'no','yes','unknown')
related with the last contact of the current campaign:
8 - contact: contact communication type (categorical: 'cellular','telephone')
9 - month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
10 - day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
11 - duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
other attributes:
12 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
13 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
14 - previous: number of contacts performed before this campaign and for this client (numeric)
15 - poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')
social and economic context attributes
16 - emp.var.rate: employment variation rate - quarterly indicator (numeric)
17 - cons.price.idx: consumer price index - monthly indicator (numeric)
18 - cons.conf.idx: consumer confidence index - monthly indicator (numeric)
19 - euribor3m: euribor 3 month rate - daily indicator (numeric)
20 - nr.employed: number of employees - quarterly indicator (numeric)

Output variable (desired target):
21 - y - has the client subscribed a term deposit? (binary: 'yes','no')

Data Exploration

Northeastern University

- Let’s first load the data set and explore its basic properties

```
df = pd.read_csv('banking.csv')
df.head()
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp_va
0	44	blue-collar	married	basic.4y	unknown	yes	no	cellular	aug	thu	...	1	999	0	nonexistent	
1	53	technician	married	unknown	no	no	no	cellular	nov	fri	...	1	999	0	nonexistent	
2	28	management	single	university.degree	no	yes	no	cellular	jun	thu	...	3	6	2	success	
3	39	services	married	high.school	no	no	no	cellular	apr	fri	...	2	999	0	nonexistent	
4	55	retired	married	basic.4y	no	yes	no	cellular	aug	fri	...	1	3	1	success	

5 rows × 21 columns

Data Exploration

Northeastern University

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
age               41188 non-null int64
job               41188 non-null object
marital           41188 non-null object
education         41188 non-null object
default           41188 non-null object
housing           41188 non-null object
loan              41188 non-null object
contact           41188 non-null object
month             41188 non-null object
day_of_week       41188 non-null object
duration          41188 non-null int64
campaign          41188 non-null int64
pdays             41188 non-null int64
previous          41188 non-null int64
poutcome          41188 non-null object
emp_var_rate      41188 non-null float64
cons_price_idx    41188 non-null float64
cons_conf_idx     41188 non-null float64
euribor3m         41188 non-null float64
nr_employed       41188 non-null float64
y                 41188 non-null int64
dtypes: float64(5), int64(6), object(10)
memory usage: 6.6+ MB
```

Data Exploration

Northeastern University

- The number of samples that belong to each class:

```
df['y'].value_counts()
```

```
0    36548
1     4640
Name: y, dtype: int64
```

- The two classes are imbalanced
- The ratio of no-subscription to subscription instances is 89:11

Data Exploration

Northeastern University

- Let's get some basic statistics on the data in each class:

```
df.groupby('y').mean()
```

y	age	duration	campaign	pdays	previous	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m	nr_employed
0	39.911185	220.844807	2.633085	984.113878	0.132374	0.248875	93.603757	-40.593097	3.811491	5176.166600
1	40.913147	553.191164	2.051724	792.035560	0.492672	-1.233448	93.354386	-39.789784	2.123135	5095.115991

- Observations:
  - Surprisingly, campaigns (number of contacts made during this campaign) are lower for customers who bought the term deposit, but number of contacts performed before this campaign are higher
  - The pdays (days since the customer was last contacted) is lower for the customers who bought the term deposit

Training and Test Split

Northeastern University

- Let's split the data set into 80% training and 20% test:

```
X = df.drop('y', axis=1)
y = df['y']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
X_train.shape
```

```
(32950, 20)
```

```
X_test.shape
```

```
(8238, 20)
```

ColumnTransformer

Northeastern University

- Many data sets contain **heterogeneous** data types, each requiring different types of data transformations
  - e.g., we may want to scale the numeric features and one-hot encode the categorical ones
- The class **ColumnTransfomer** in sklearn.compose allows you to apply different transformers to different columns of an array or a DataFrame

> *class* `sklearn.compose.` **`ColumnTransformer`**(*transformers*, *, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*) [source]

- List of (name, transformer, columns) tuples specify the transformer objects to be applied to subsets of the data

ColumnTransformer

Northeastern University

- For example, let's define a transformer to normalize the numerical features, and one-hot encode the categorical features

```
from sklearn.compose import ColumnTransformer

categorical_features = list(X.dtypes[X.dtypes == 'object'].index)
numerical_features = list(X.dtypes[X.dtypes != 'object'].index)

transformer = ColumnTransformer([
    ('scaler', StandardScaler(), numerical_features),
    ('onehot', OneHotEncoder(), categorical_features)
])
```

- Then we define a pipeline that combines the transformer with logistic regression:

```
clf = Pipeline([
    ('trans', transformer),
    ('log_reg', LogisticRegression())
])
```

Performance Measures

Northeastern University

- Training the classifier on the data set provides the following results:

```
clf.fit(X_train, y_train)

print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9112
Test set accuracy: 0.9139
```

- Above 91% accuracy (ratio of correct predictions)
- Looks good, no?

Performance Measures

Northeastern University

- Let's create a dumb classifier that just classifies every sample as 0 (no subscription)

```
from sklearn.base import BaseEstimator
from sklearn.metrics import get_scorer

class DumbClassifier(BaseEstimator):
    def fit(self, X, y):
        pass
    def predict(self, X):
        return np.zeros((X.shape[0], 1))
    def score(self, X, y):
        scoring = get_scorer('accuracy')
        return scoring(self, X, y)

d_clf = DumbClassifier()
d_clf.fit(X_train, y_train)
print('Training set accuracy:', np.round(d_clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(d_clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.8869
Test set accuracy: 0.8892
```

- This classifier has about 89% accuracy!

We need better evaluation metrics

Confusion Matrix

Northeastern University

- A **confusion matrix** is a table that summarizes the number of instances predicted correctly or incorrectly by a classification model

		Predicted Class	
		+	−
Actual Class	+	f_{++} (TP)	f_{+-} (FN)
	−	f_{-+} (FP)	f_{--} (TN)

  - **True positive** (TP) – the number of examples correctly identified as positive
  - **True negative** (TN) – the number of examples correctly identified as negative
  - **False positive** (FP) – the number of examples wrongly classified as positive
  - **False negative** (FN) – the number of examples incorrectly identified as negative

Confusion Matrix

Northeastern University

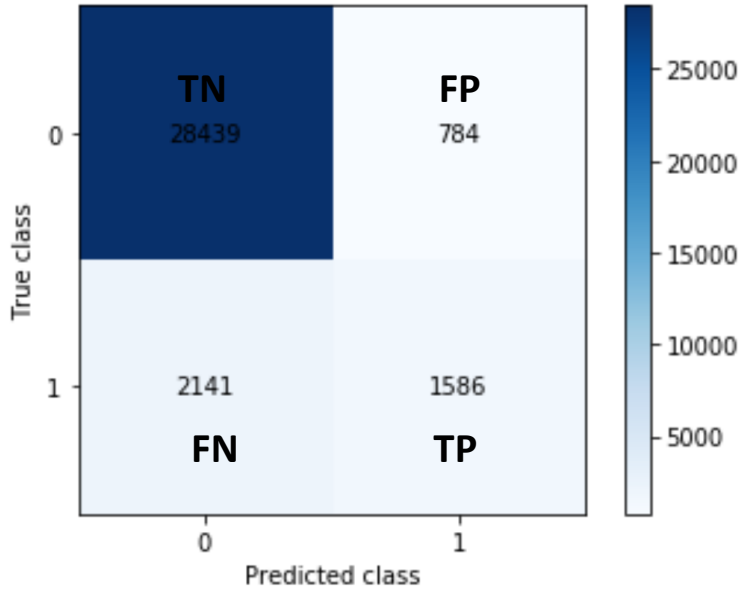
- To compute the confusion matrix, we can use the function **confusion_matrix()**:

```
from sklearn.metrics import confusion_matrix

y_pred = clf.predict(X_train)
conf_mat = confusion_matrix(y_train, y_pred)
print(conf_mat)
```

```
[[28439   784]
 [ 2141  1586]]
```

```
def plot_confusion_matrix(conf_mat):
    n = len(conf_mat)
    plt.imshow(conf_mat, cmap='Blues', extent=[-0.5, n-0.5, -0.5, n-0.5])
    for i in range(n):
        for j in range(n):
            plt.text(i, j, conf_mat[n-j-1, i], ha='center', va='center')
    plt.colorbar()
    plt.xticks(range(n))
    plt.yticks(range(n), range(n-1, -1, -1))
    plt.xlabel('Predicted class')
    plt.ylabel('True class');
```

```
plot_confusion_matrix(conf_mat)
```



Precision and Recall

Northeastern University

- **Precision** is the fraction of true positive samples in the group of samples declared as positive by the classifier

$$\text{precision} = \frac{TP}{TP + FP}$$

- **Recall** is the fraction of positive examples that are correctly classified by the model

$$\text{recall} = \frac{TP}{TP + FN}$$

- Improving precision typically reduces recall and vice versa
  - e.g., a model that classifies every sample as positive will have a perfect recall, but very poor precision

Precision and Recall

Northeastern University

- Computing the precision and recall using Scikit-Learn:

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_train, y_pred)
```

```
0.6691983122362869
```

```
recall_score(y_train, y_pred)
```

```
0.4255433324389589
```

- Now our model doesn't look as shiny as it did when we looked at its accuracy
  - When it claims a customer subscribed, it is correct only 67.2% of the time
  - Moreover, it only detects 43.6% of the subscriptions

F_1 Score

Northeastern University

- Precision and recall can be combined into a single metric known as F_1 score
- F_1 score is the **harmonic mean** of precision and recall:

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

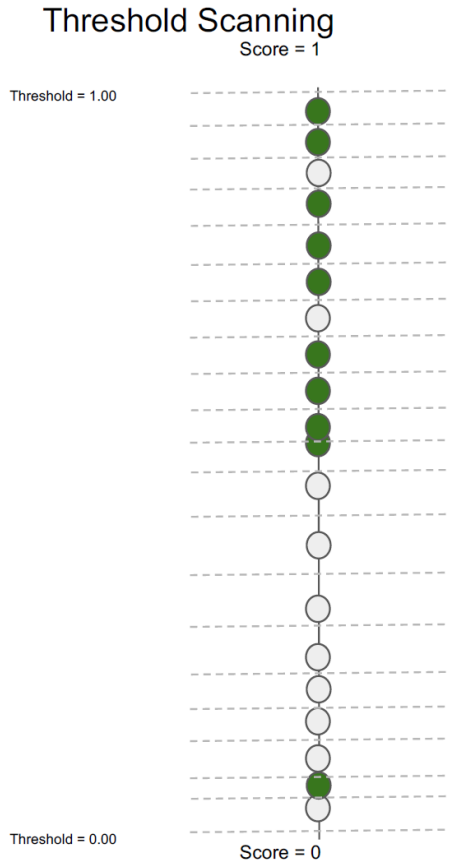
- The harmonic mean of two numbers tends to be closer to the smaller of the two
- Hence, a high F_1 score ensures that both precision and recall are reasonably high
- To compute the F_1 score, simply call the **f1_score()** function:

```
from sklearn.metrics import f1_score
f1_score(y_train, y_pred)
```

```
0.5202558635394456
```

Thresholding

- Many classifiers let you specify the **decision threshold**, which is the value above which the classifier assigns the example to the positive class



Threshold	TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	F1
1.00	0	10	0	10	0.50	1	0	1	0
0.95	1	10	0	9	0.55	1	0.1	1	0.182
0.90	2	10	0	8	0.60	1	0.2	1	0.333
0.85	2	9	1	8	0.55	0.667	0.2	0.9	0.308
0.80	3	9	1	7	0.60	0.750	0.3	0.9	0.429
0.75	4	9	1	6	0.65	0.800	0.4	0.9	0.533
0.70	5	9	1	5	0.70	0.833	0.5	0.9	0.625
0.65	5	8	2	5	0.65	0.714	0.5	0.8	0.588
0.60	6	8	2	4	0.70	0.750	0.6	0.8	0.667
0.55	7	8	2	3	0.75	0.778	0.7	0.8	0.737
0.50	8	8	2	2	0.80	0.800	0.8	0.8	0.800
0.45	9	8	2	1	0.85	0.818	0.9	0.8	0.857
0.40	9	7	3	1	0.80	0.750	0.9	0.7	0.818
0.35	9	6	4	1	0.75	0.692	0.9	0.6	0.783
0.30	9	5	5	1	0.70	0.643	0.9	0.5	0.750
0.25	9	4	6	1	0.65	0.600	0.9	0.4	0.720
0.20	9	3	7	1	0.60	0.562	0.9	0.3	0.692
0.15	9	2	8	1	0.55	0.529	0.9	0.2	0.667
0.10	9	1	9	1	0.50	0.500	0.9	0.1	0.643
0.05	10	1	9	0	0.55	0.526	1	0.1	0.690
0.00	10	0	10	0	0.50	0.500	1	0	0.667

Thresholding

Northeastern University

- Many classifiers provide the decision scores used to make their predictions via the function **decision_function()**

```
y_scores = clf.decision_function(X_train[10:20])
y_scores
```

```
array([-4.02670565, -4.30432168,  0.31334842, -0.28544069, -1.08385411,
       -3.89626759,  1.40809097, -4.65313728, -1.00499063,  0.61793247])
```

```
y_pred[10:20]
```

```
array([0, 0, 1, 0, 0, 0, 1, 0, 0, 1], dtype=int64)
```

- In logistic regression, the decision score represents the signed distance of the sample from the separating hyperplane
- A threshold of 0 is used to predict if the sample belongs to the positive or negative class

Thresholding

Northeastern University

- We can change the threshold in order to increase the precision/recall at the expense of the other one
- For example, we can lower the threshold from 0 to -1.0:

```
y_scores = clf.decision_function(X_train)
threshold = -1.0
new_y_pred = (y_scores > threshold)
```

```
precision_score(y_train, new_y_pred)
```

```
0.5700045310376076
```

```
recall_score(y_train, new_y_pred)
```

```
0.6750737858867722
```

Precision-Recall Curve

Northeastern University

- How can you decide which threshold to use?
- A **precision-recall curve** shows the tradeoff between precision and recall for different thresholds
- To generate this curve we first compute the precisions and recalls for all possible thresholds using the function precision_recall_curve():

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train, y_scores)
```

```
precisions
```

```
array([0.11373554, 0.1137085 , 0.11371197, ..., 0.        , 0.        ,
       1.        ])
```

```
recalls
```

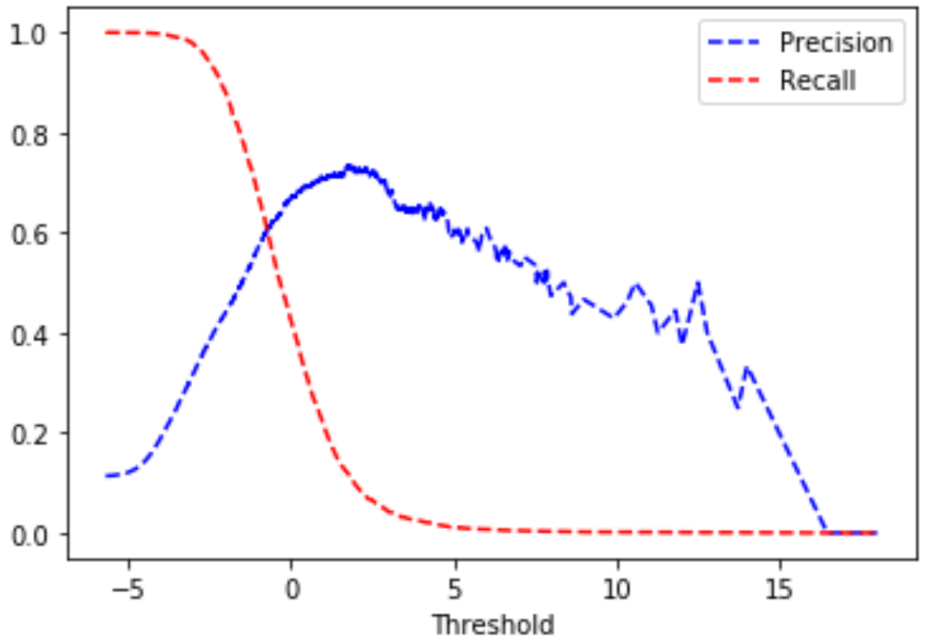
```
array([1.        , 0.99973169, 0.99973169, ..., 0.        , 0.        ,
       0.        ])
```

Precision-Recall Curve

Northeastern University

- We can now plot the precision and recall as functions of the threshold value:

```
plt.plot(thresholds, precisions[:-1], 'b--', label='Precision')
plt.plot(thresholds, recalls[:-1], 'r--', label='Recall')
plt.xlabel('Threshold')
plt.legend();
```



Precision-Recall Curve

Northeastern University

- Let's suppose we decide to aim for 80% recall
- We can use the recalls array to find which threshold we need to use:

```
threshold = thresholds[np.where(recalls > 0.8)][-1]
threshold
```

```
-1.5731546420376081
```

```
y_recall_80 = (y_scores > threshold)
precision_score(y_train, y_recall_80)
```

```
0.4906995884773663
```

```
recall_score(y_train, y_recall_80)
```

```
0.7998390126106788
```

  - Precision decreased from 57% to 49% while recall increased from 67% to 80%

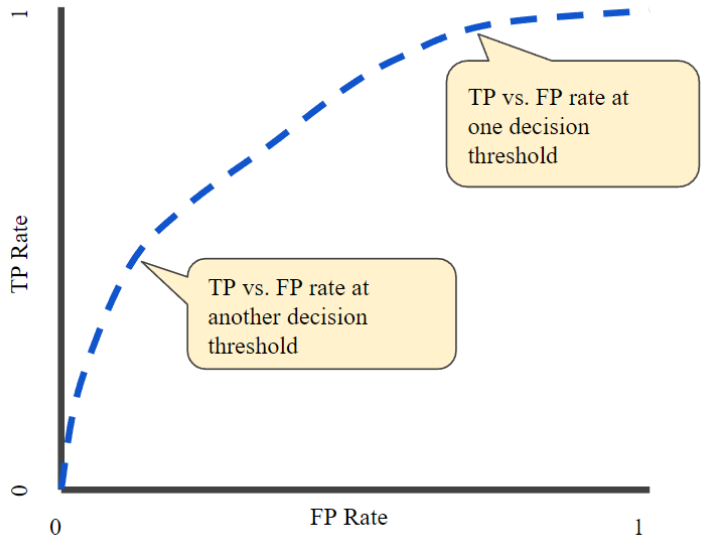
ROC Curve

Northeastern University

- A graph showing the performance of a classifier at different thresholds
  - The true positive rate (TPR) = recall is plotted along the *y* axis
    - The fraction of the positive samples that were classified as positive
  - The false positive rate (FPR) is shown on the *x* axis
    - The fraction of the negative samples that were classified as positive

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{TN + FP}$$



- Lower decision threshold ⇒ more samples are classified as positive ⇒ both TPR and FPR increase
- The ideal model has TPR = 1, FPR = 0
- A model that makes random guesses resides along the main diagonal (TPR = FPR)

ROC Curve

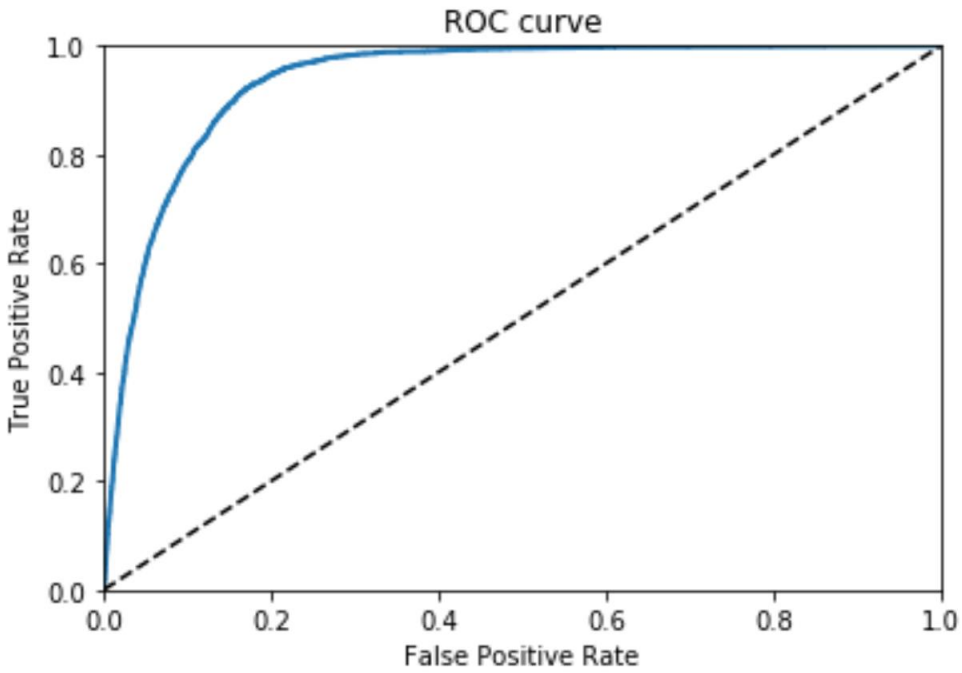
Northeastern University

- In Scikit-learn you can use the **roc_curve()** function:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train, y_scores)
```

```
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, lw=2)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve')
```

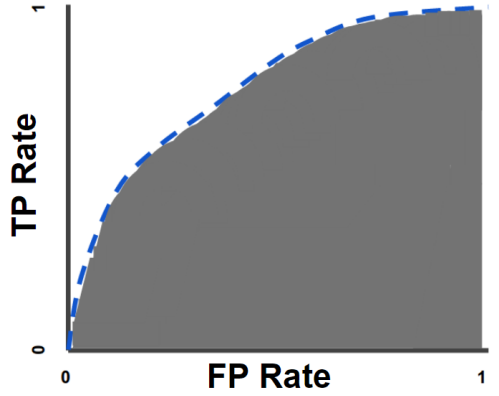
```
plot_roc_curve(fpr, tpr)
```



AUC

Northeastern University

- AUC = area under the ROC curve
- Used to evaluate which model is better on average
- A perfect classifier will have AUC = 1
- A classifier that makes random guesses will have AUC = 0.5



- Computing the AUC in Scikit-learn:

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_train, y_scores)
```

```
0.9363915446739913
```

Sampling-Based Approaches

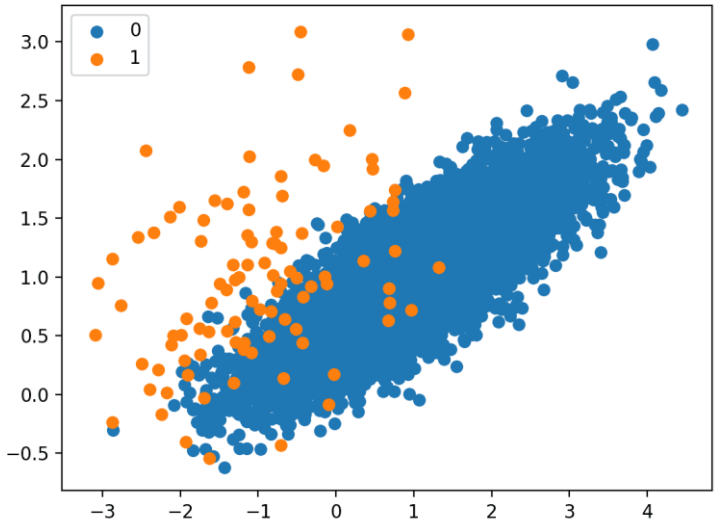
Northeastern University

- Sampling is another approach for handling the class imbalance problem
- **Idea**: modify the distribution of samples so that the rare class is well represented in the training set
- For example, assume we have a training set with 100 positive examples and 1000 negative examples
- **Undersampling** – a random sample of 100 negative examples is chosen to form the training set along with all the positive examples
  - We may remove negative examples that are far from the decision boundary
- **Oversampling** – replicate the positive examples until the training set has an equal number of positive and negative examples

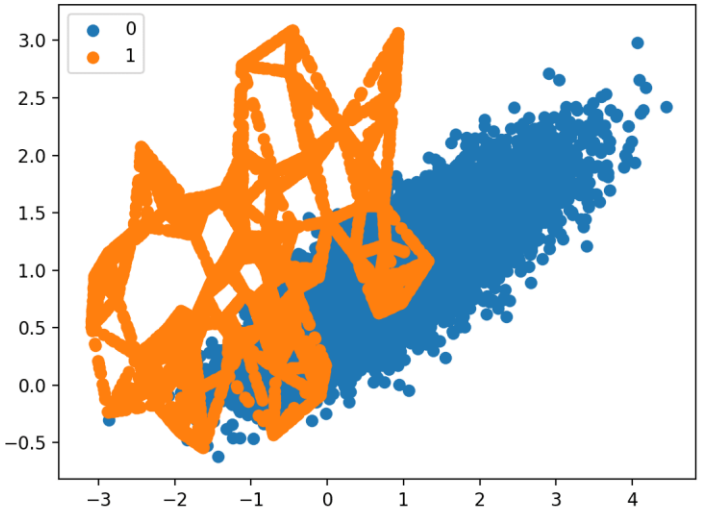
SMOTE

Northeastern University

- Synthesize new examples from the existing examples in the minority class
  - Choose a random example from the minority class
  - Find *k* of the nearest neighbors for that example (typically *k* = 5)
  - Select a random neighbor
  - Create a new example at a randomly selected point between the two examples



Before SMOTE



After SMOTE

SMOTE

Northeastern University

- The python library imbalanced-learn implements this technique
- Example for using SMOTE with the banking data set

```
from imblearn.over_sampling import SMOTE
```

```
sm = SMOTE()

X_trans = transformer.fit_transform(X)
X_sm, y_sm = sm.fit_resample(X_trans, y)
```

```
print(f'Shape of X before SMOTE: {X.shape}')
print(f'Shape of X after SMOTE: {X_sm.shape}')

print('Balance of positive and negative classes (%):')
y_sm.value_counts(normalize=True) * 100
```

```
Shape of X before SMOTE: (41188, 20)
Shape of X after SMOTE: (73096, 63)
Balance of positive and negative classes (%):

1    50.0
0    50.0
Name: y, dtype: float64
```

SMOTE

Northeastern University

- Results on the test set after using SMOTE

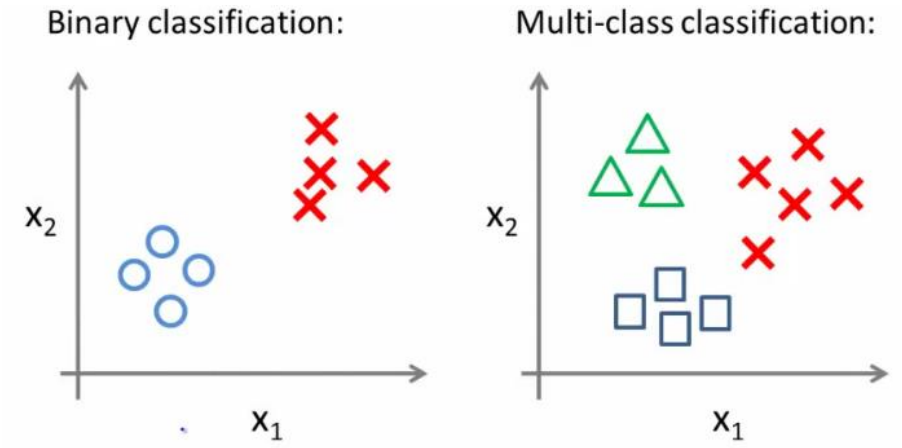
```
X_train, X_test, y_train, y_test = train_test_split(X_trans, y, test_size=0.2)

y_pred_test = model.predict(X_test)

print(f'Accuracy (test) = {model.score(X_test, y_test):.3f}')
print(f'Precision (test) = {precision_score(y_test, y_pred_test):.3f}')
print(f'Recall (test) = {recall_score(y_test, y_pred_test):.3f}')
print(f'F1 score (test) = {f1_score(y_test, y_pred_test):.3f}')
```

```
Accuracy (test) = 0.864
Precision (test) = 0.456
Recall (test) = 0.879
F1 score (test) = 0.600
```

Multiclass Problems

Northeastern University

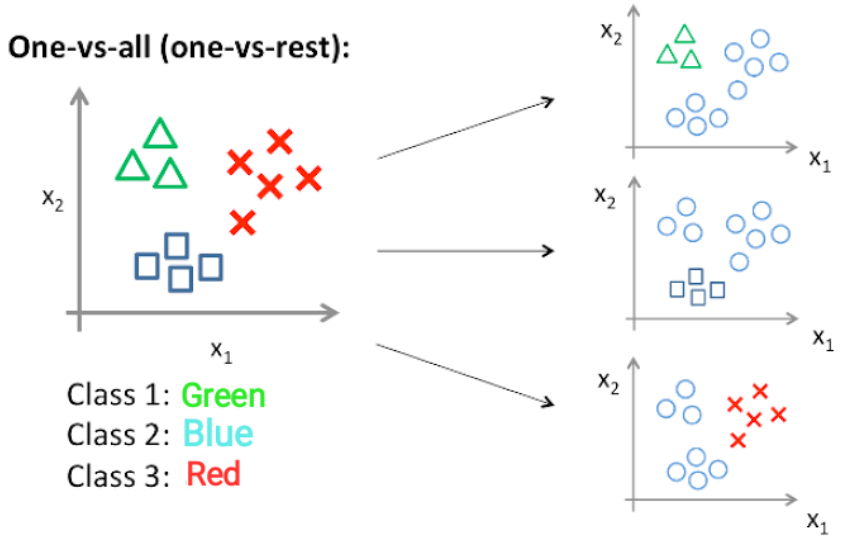
- In many real-world problems the data is divided to more than two categories
  - e.g., face recognition, text classification
- Some ML algorithms are originally designed to handle only binary classification
  - e.g., logistic regression, SVM
- There are several approaches to extend binary classifiers into multiclass classifiers



One vs. Rest (OvR)

Northeastern University

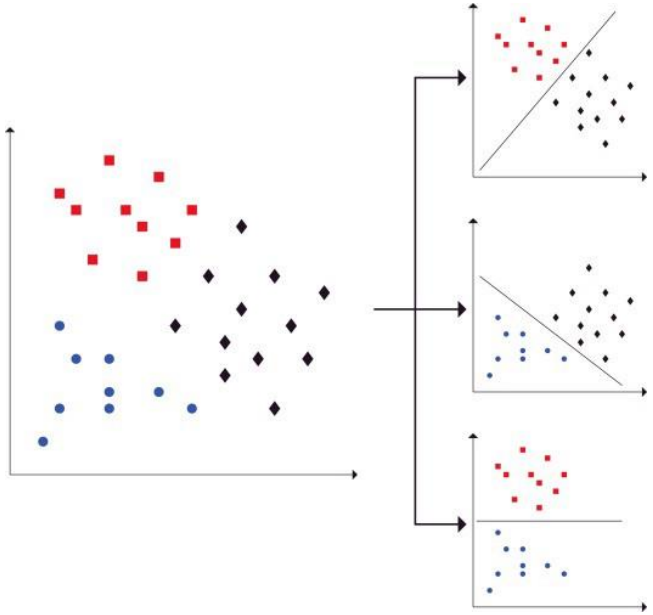
- Train a binary classifier per class, with the samples of that class as positive samples and all other samples as negatives
- The base classifiers should provide a real-valued confidence score for their decision
- Predict the label *k* for which the class had the highest confidence score
- Issues
  - The scale of the confidence scores may differ
  - Each classification problem is imbalanced



One vs. One (OvO)

Northeastern University

- Each binary classifier is used to distinguish between a pair of classes (c_i, c_j)
- Samples that don't belong to either c_i or c_j are ignored
- Use majority voting to make the final prediction
- The number of classifiers required is $\binom{K}{2} = \frac{K(K-1)}{2}$



- Advantages of OvO:
  - Usually provides better results than OvR
    - Each individual learning problem involves a small subset of the data
- Disadvantages of OvO:
  - Slower than OvR due to its $O(K^2)$ complexity
  - Doesn't provide probability estimates for the different classes

Example: MNIST

Northeastern University

- To demonstrate multi-class classification, we'll use the MNIST dataset
- This is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau
- Each image is labeled with the digit it represents
- This data set has been studied so much that it is often called the "Hello World" of Machine Learning
- Description of the data set can be found here

Example: MNIST

Northeastern University

- The following code loads the MNIST dataset using sklearn.datasets.fetch_openml

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('mnist_784', return_X_y=True)
```

```
X.shape
```

```
(70000, 784)
```

```
y.shape
```

```
(70000,)
```

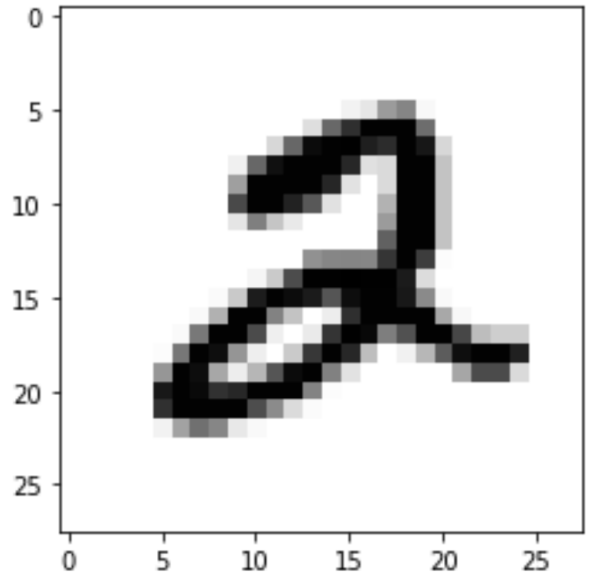
- There are 70,000 images of size 28×28, i.e., each image has 784 features
- Each feature represents one pixel's intensity from 0 (white) to 255 (black)

Example: MNIST

Northeastern University

- Let's take a peek at one digit from the data set:

```
digit_index = 5
digit_image = X[digit_index].reshape(28, 28)
plt.imshow(digit_image, cmap='binary');
```




```
y[digit_index]
```

```
'2'
```

Example: MNIST

Northeastern University

- Let's examine how many images we have from each digit type:

```
np.unique(y, return_counts=True)
```

```
(array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object),
 array([6903, 7877, 6990, 7141, 6824, 6313, 6876, 7293, 6825, 6958],
       dtype=int64))
```

  - The data set is well balanced
- The first 60,000 images belong to the training set and the last 10,000 to the test set
- So we can split the data set into training and test sets simply by:

```
X_train, y_train = X[:60000], y[:60000]
X_test, y_test = X[60000:], y[60000:]
```

OvR Classification

Northeastern University

- For OvR classification, we can wrap our classifier with an OneVsRestClassifier object

class sklearn.multiclass.**OneVsRestClassifier**(*estimator*, *, *n_jobs=None*) [source]

- Let's train a logistic regression model on MNIST using OvR strategy:

```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

clf = OneVsRestClassifier(LogisticRegression(), n_jobs=-1)
clf.fit(X_train, y_train)
```

```
OneVsRestClassifier(estimator=LogisticRegression(), n_jobs=-1)
```

- The number of estimators created is:

```
len(clf.estimators_)
```

```
10
```

OvR Classification

Northeastern University

- The accuracy of the classifier is:

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9261
Test set accuracy: 0.9181
```

- We can use it to predict the label for a given image:

```
clf.predict([X[digit_index]])
```

```
array(['2'], dtype='<U1')
```

```
np.round(clf.predict_proba([X[digit_index]]), 3)
```

```
array([[0.005, 0.   , 0.971, 0.   , 0.   , 0.002, 0.   , 0.   , 0.017,
        0.004]])
```

OvO Classification

Northeastern University

- For OvO classification, we can wrap our classifier with an OneVsOneClassifier object:

```
from sklearn.multiclass import OneVsOneClassifier

clf = OneVsOneClassifier(LogisticRegression(), n_jobs=-1)
clf.fit(X_train, y_train)
```

```
OneVsOneClassifier(estimator=LogisticRegression(), n_jobs=-1)
```

- The number of estimators created is:

```
len(clf.estimators_)
```

```
45
```

- The accuracy scores are:

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9734
Test set accuracy: 0.9278
```

Softmax Regression

Northeastern University

- **Softmax regression** (or **multinomial logistic regression**) is a generalization of logistic regression to multiclass problems
- Given a sample input **x**, we estimate the probabilities for each class *k* (*k* = 1, …, *K*)

$$p_i = P(y = i|\mathbf{x}) \qquad \sum_{i=1}^{k} p_i = 1$$

- Similar to logistic regression, we assume that the log odds between each probability to some reference probability (e.g., p_k) is a linear combination of the features:

$$\frac{p_i}{p_k} = e^{\mathbf{w}_i^T \mathbf{x}}$$

$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}}$$

- Note that there are *K* vectors of weights that need to be learned from the data

Softmax Regression

Northeastern University

- Since all the probabilities sum to 1, we have:

$$p_k = 1 - \sum_{i=1}^{k-1} p_i = 1 - \sum_{i=1}^{k-1} \left(p_k e^{\mathbf{w}_i^T \mathbf{x}} \right)$$

$$p_k + p_k \left(\sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$

$$p_k \left(1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$

$$p_k = \frac{1}{1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}}}$$

- Substituting back to p_i we get:

$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}} = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{1 + \sum_{j=1}^{k-1} e^{\mathbf{w}_j^T \mathbf{x}}}, \qquad 1 \le i < k$$

- By choosing arbitrarily $\mathbf{w}_k = 0$:

$$p_i = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{\sum_{j=1}^{k} e^{\mathbf{w}_j^T \mathbf{x}}}, \qquad 1 \le i \le k$$

The Softmax Function

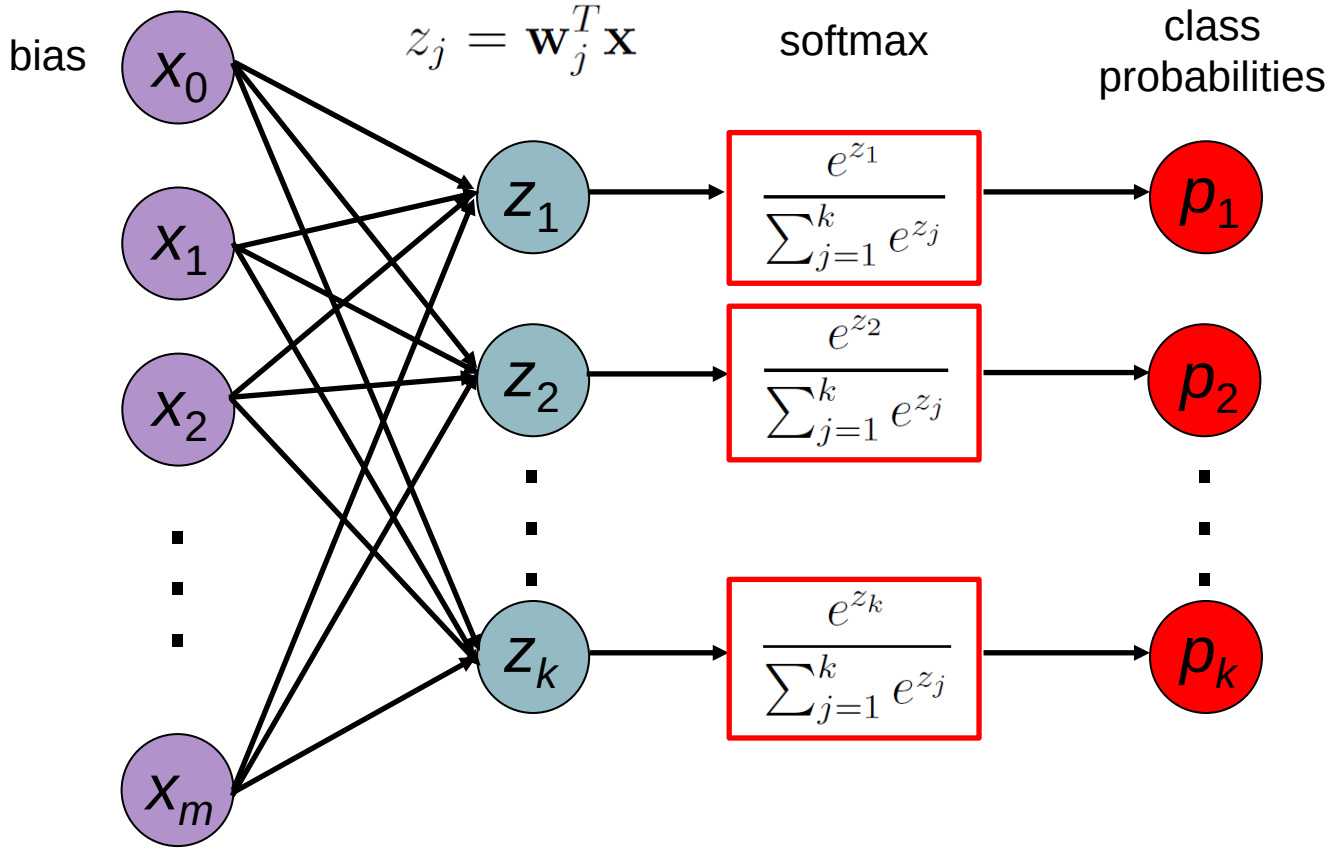
Northeastern University

- The **softmax function**

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Smoothed version of the max function
- Example: the softmax of (1, 2, 6) is (0.007, 0.018, 0.976)
- The sigmoid function is a special case of the softmax function for a classifier with only two input classes

Softmax Regression



Maximum Likelihood Estimation

Northeastern University

- Next, we write the likelihood function
- From our assumption we can write

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{k=1}^{K} p(C_k|\mathbf{x})^{\mathbf{y}_k} = \prod_{k=1}^{K} (h_k(\mathbf{x}))^{\mathbf{y}_k}$$

- The likelihood function is then given by

$$\mathcal{L}(\mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{i=1}^{n} p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{i=1}^{n} \prod_{k=1}^{K} (h_k(\mathbf{x}_i))^{y_{ik}}$$

- Taking the negative logarithm gives

$$J(\mathbf{w}_1, ..., \mathbf{w}_K) = -\log \mathcal{L}(\mathbf{w}_1, ..., \mathbf{w}_K) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(h_k(\mathbf{x}_i))$$

- This function is known as the **cross-entropy loss function**

Cross-Entropy Loss

Northeastern University

- Extension of the log loss to the multi-class case
- Given a vector of estimated probabilities **p** and a vector **y** that represents a one-hot encoding of the label, the cross-entropy loss is defined as

$$L_{\mathrm{CE}}(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{k} y_i \log p_i$$

- Example: assume we have a three-class problem (k = 3) and a sample whose true label is class 2 ($\mathbf{y} = (0, 1, 0)^T$), and the model's prediction is $\mathbf{p} = (0.3, 0.6, 0.1)^T$
- Then the cross-entropy loss for that sample is:

$$L_{\mathrm{CE}} = -(0 \cdot \log 0.3 + 1 \cdot \log 0.6 + 0 \cdot \log 0.1) = 0.5108$$

Softmax Regression in Scikit-Learn

Northeastern University

- The LogisticRegression class supports the cross-entropy loss

class `sklearn.linear_model.` **`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) [source]

- The multi_class argument can have one of the following options:
  - 'ovr' – uses the one-vs-rest (OvR) scheme
  - 'multinomial' – uses the cross-entropy loss
  - 'auto' (the default) selects 'ovr' if the data is binary, or if solver='liblinear' (which doesn't support multinomial), and otherwise selects 'multinomial'

Example: MNIST

Northeastern University

- Let's use softmax regression to classify the MNIST digits

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

```
print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
```

```
Training set accuracy: 0.9339
Test set accuracy: 0.9255
```

- We got similar results to OvO, but this time using a single classifier

Example: MNIST

Northeastern University

- Comparing between the different solvers:

```
for solver in ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']:
    clf = LogisticRegression(solver=solver)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()

    print('Solver:', solver)
    print('Training set accuracy:', np.round(clf.score(X_train, y_train), 4))
    print('Test set accuracy:' , np.round(clf.score(X_test, y_test), 4))
    print('Elpased time:', np.round(end - start, 3), 'seconds')
```

```
Solver: lbfgs
Training set accuracy: 0.9339
Test set accuracy: 0.9255
Elpased time: 13.784 seconds
```

```
Solver: liblinear
Training set accuracy: 0.9312
Test set accuracy: 0.9171
Elpased time: 2490.652 seconds
```

```
Solver: saga
Training set accuracy: 0.9386
Test set accuracy: 0.9257
Elpased time: 203.123 seconds
```

```
Solver: newton-cg
Training set accuracy: 0.9448
Test set accuracy: 0.9208
Elpased time: 1524.278 seconds
```

```
Solver: sag
Training set accuracy: 0.9396
Test set accuracy: 0.9255
Elpased time: 153.617 seconds
```

Classification Metrics in Multiclass Problems

Northeastern University

- Confusion matrix will be $K \times K$
- Most metrics (except accuracy) are analyzed as multiple 1-vs-many
- A **macro-average** computes the metric independently for each class and then takes the average (hence treating all classes equally)
- A **micro-average** aggregates the contributions of all classes to compute the average
- For example, for the precision measure

$$\text{Macro-Precision} = \frac{\sum_{k=1}^{K} \text{Precision}_k}{K}$$

$$\text{Micro-Precision} = \frac{\sum_{k=1}^{K} TP_k}{\sum_{k=1}^{K} TP_k + \sum_{k=1}^{K} FP_k}$$

- Micro-average is preferable if you suspect there might be a class imbalance

Classification Metrics in Multiclass Problems

Northeastern University

- For multi-class problems, you can use sklearn.metrics.classification_report() to show a summary of the precision, recall, and F1 score for each class:

```
from sklearn.metrics import classification_report

y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

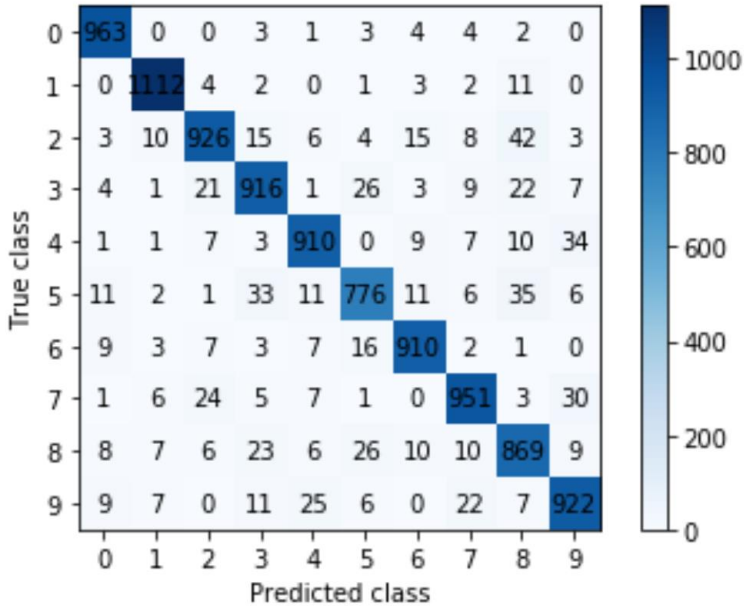
```
              precision    recall  f1-score   support

           0       0.95      0.98      0.97       980
           1       0.97      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.90      0.91      0.91      1010
           4       0.93      0.93      0.93       982
           5       0.90      0.87      0.89       892
           6       0.94      0.95      0.95       958
           7       0.93      0.93      0.93      1028
           8       0.87      0.89      0.88       974
           9       0.91      0.91      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.93      0.93      0.93     10000
```

Error Analysis

Northeastern University

- Let's examine the confusion matrix

```
from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(y_test, y_pred)
plot_confusion_matrix(conf_mat)
```

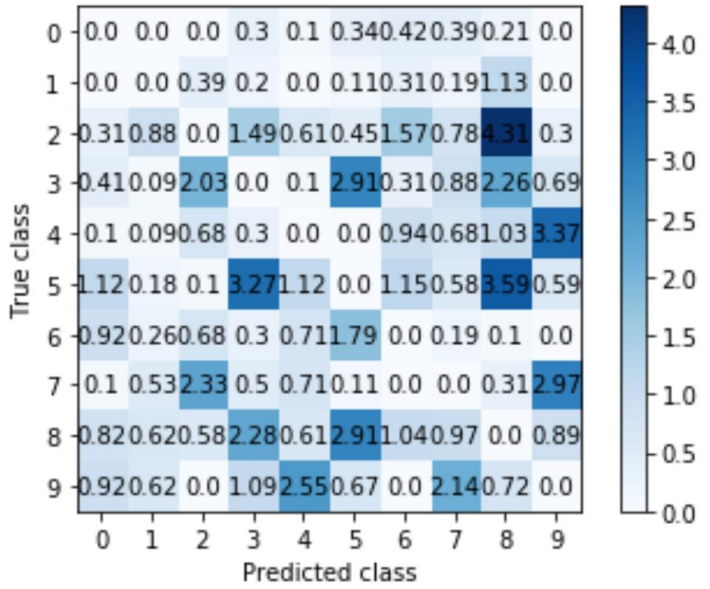


- Since the number of digits in each class are not equal, it makes more sense to examine the relative errors instead of the absolute

Error Analysis

- To make the errors in the confusion matrix be relative instead of absolute:
  - Divide each value in the matrix by the number of images in the corresponding class
  - Fill the diagonal with zeros to keep only the errors

```
norm_conf_mat = conf_mat / conf_mat.sum(axis=1) * 100
np.fill_diagonal(norm_conf_mat, 0)
norm_conf_mat = np.round(norm_conf_mat, 2)

plot_confusion_matrix(norm_conf_mat)
```



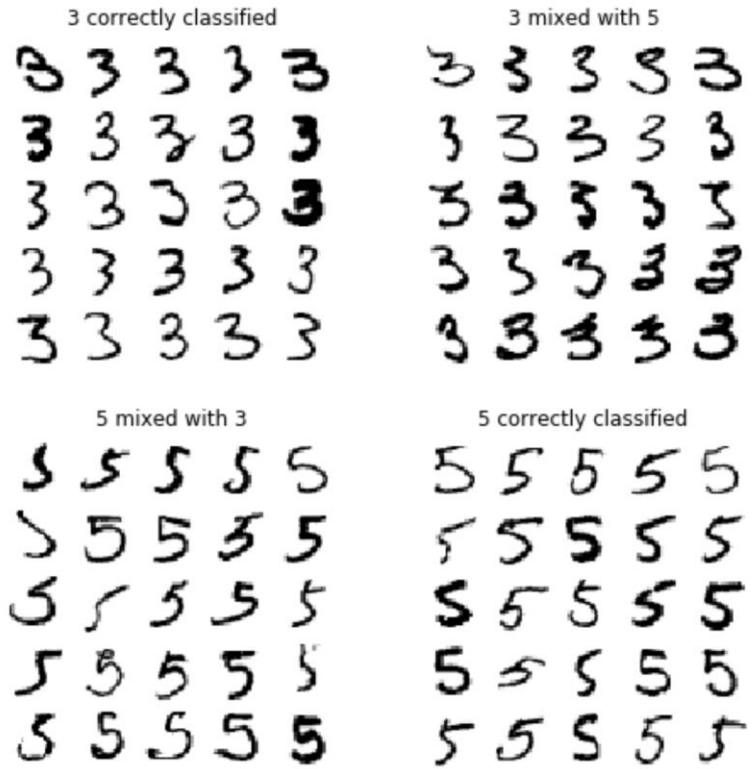
- It seems that our main confusions are between 3/5, 4/9, 5/8, and 7/9

Error Analysis

Northeastern University

- Analyzing individual errors can help you gain insights on what your classifier is doing
- For example, we can plot examples of 3s and 5s from the test set:

```
cls_a, cls_b = '3', '5'
X_aa = X_test[(y_test == cls_a) & (y_pred == cls_a)]
X_ab = X_test[(y_test == cls_a) & (y_pred == cls_b)]
X_ba = X_test[(y_test == cls_b) & (y_pred == cls_a)]
X_bb = X_test[(y_test == cls_b) & (y_pred == cls_b)]

fig, ax = plt.subplots(2, 2, figsize=(8, 8))
plot_digits(ax[0, 0], X_aa[:25], images_per_row=5,
            title='3 correctly classified')
plot_digits(ax[0, 1], X_ab[:25], images_per_row=5,
            title='3 mixed with 5')
plot_digits(ax[1, 0], X_ba[:25], images_per_row=5,
            title='5 mixed with 3')
plot_digits(ax[1, 1], X_bb[:25], images_per_row=5,
            title='5 correctly classified')
```



Logistic Regression Summary

Northeastern University

Pros

- Efficient to train
- Easy to implement
- Provides class probabilities
- Won't overfit easily, as it's a linear model
- Highly interpretable
  - there is a different weight for each feature
- Can handle irrelevant/redundant features
  - e.g., by assigning weights close to 0 for them
- A small number of hyperparameters

Cons

- Can learn only linear decision boundaries
- Typically outperformed by more complex algorithms
- Cannot handle missing values
- Requires scaling of the data